

Variations on Multi-Core Nested Depth-First Search

Alfons Laarman

Jaco van de Pol

Formal Methods and Tools, University of Twente, The Netherlands

{a.w.laarman,vdpol}@cs.utwente.nl

Recently, two new parallel algorithms for on-the-fly model checking of LTL properties were presented at the same conference: *Automated Technology for Verification and Analysis, 2011*. Both approaches extend Swarmed NDFS, which runs several sequential NDFS instances in parallel. While parallel random search already speeds up detection of bugs, the workers must share some global information in order to speed up full verification of correct models. The two algorithms differ considerably in the global information shared between workers, and in the way they synchronize.

Here, we provide a thorough experimental comparison between the two algorithms, by measuring the runtime of their implementations on a multi-core machine. Both algorithms were implemented in the same framework of the model checker LTSMIN, using similar optimizations, and have been subjected to the full BEEM model database.

Because both algorithms have complementary advantages, we constructed an algorithm that combines both ideas. This combination clearly has an improved speedup. We also compare the results with the alternative parallel algorithm for accepting cycle detection OWCTY-MAP. Finally, we study a simple statistical model for input models that do contain accepting cycles. The goal is to distinguish the speedup due to parallel random search from the speedup that can be attributed to clever work sharing schemes.

1 Introduction

Model checking is an important technique to automatically verify that a system's behavior is free from subtle bugs, for instance violations of safety and liveness requirements. Linear Time Logic (LTL) expresses such requirements as properties on individual runs of a system. LTL model checking reduces to detecting accepting cycles in a so-called Büchi automaton [20]. A linear-time algorithm to detect those cycles is the Nested Depth-First Search (NDFS) algorithm, introduced by Courcoubetis et al. [6]. NDFS can also terminate as soon as some accepting cycle is found, which makes it very useful for bug hunting. Still, model checking is a time- and memory-consuming procedure due to the sheer size of the state space of realistic systems, leading to an extremely large Büchi automaton.

During the last decades, processor speeds have been greatly increased, making model checkers much more powerful. Where early papers discussed the verification of models with a few thousand states, currently we can easily handle billions of states [16, 15, 14]. Recently, however, these advances are grinding to a halt, because of physical limits inside the CPU cores. Instead, the number of logical computing cores increases. Nonetheless, model checking can still benefit from the progress made by CPU manufacturers, if the algorithms are parallelized.

A complication is that DFS (and thus NDFS) is inherently sequential [18]. Barnat et al. have therefore introduced breadth-first search (BFS) based algorithms, such as *Maximal-Accepting-Predecessors* (MAP [5]) and *One-Way-Catch-Them-Young* (OWCTY [21]). These algorithms deliver excellent speedups, but sacrifice linear-time complexity. However, their latest combined OWCTY-MAP algorithm [4], is linear-time for the class of weak LTL properties and also useful for bug hunting. It is therefore the current state of the art in multi-core LTL model checking.

Recently, also two parallel NDFS-based algorithms were introduced [7, 13]. Both take as starting point a randomized parallel search by a swarm of NDFS workers. While this is useful for bug-hunting, it does not really help in the absence of bugs, in which case all workers traverse the full state space. To improve speedup, both algorithms share some global information between workers, in order to reduce the amount of work even in the absence of accepting cycles. ENDFS from Evangelista et al. [7] shares a lot of information, but this may break the required DFS order. A sequential repair procedure steps in when a potentially dangerous situation is detected. On the other hand, LNDFS from Laarman et al. [13] shares less global information and adds extra synchronization. This avoids dangerous situations and the need for a repair strategy. However, this leads to a reduced amount of work sharing in some cases.

Contributions. The main goal of this paper is to experimentally compare both multi-core NDFS algorithms. In order to enable a fair comparison, we extended ENDFS with the same optimizations as used in LNDFS. We implemented both algorithms in the same framework of LTSMIN. Finally, we subjected both implementations to the full BEEM benchmark database [17], running them on shared memory machines with up to 16 cores. Note that actual runtimes had not yet been reported for ENDFS, although workload distributions were shown in [7]. Also, for LNDFS, we have rerun the experiments from [13].

Another contribution is a simple combination of the ENDFS and LNDFS algorithms, improving the speedup compared to both of them. We also compare all mentioned algorithms with the OWCTY-MAP algorithm, both for bug hunting and for full verification. Finally, based on a simple statistical model [12], we investigate how much of the speedup in the parallel NDFS algorithms should be contributed to the effects of parallel random search and what is the contribution of the more clever work sharing schemes.

The algorithms are explained in Section 2. The experimental results are presented in Section 3. Section 4 contains the discussion on parallel random search. Our conclusions are summarized in Section 5.

2 Parallel Algorithms to Detect Accepting Cycles

Model checking properties from Linear Temporal Logic (LTL) entails verifying that all runs of a given system satisfy some safety or liveness property. In the automata-theoretic approach [20, 1], a Büchi automaton is constructed that accepts all infinite words corresponding to those runs of the original system that violate the property. So the problem is reduced to the emptiness check of ω -regular languages. A Büchi automaton accepts a word if it visits some accepting state infinitely often. For finite automata, this implies that there is a cycle through some accepting state.

Definition 1 A Büchi automaton is a quadruple $\mathcal{B} = (\mathcal{S}, s_I, \text{post}, \mathcal{A})$, where \mathcal{S} is the finite set of states, $s_I \in \mathcal{S}$ is the initial state, $\text{post} : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ the successor function, and $\mathcal{A} \subseteq \mathcal{S}$ the set of accepting states.

Note, that the use of the post function reflects the way in which the Büchi automaton is computed *on-the-fly* from the input model. When appropriate, we refer to the complete automaton as graph or state space.

The purpose of all algorithms in this paper is to detect an accepting cycle in this graph. For states $s, t \in \mathcal{S}$, we write $s \rightarrow t$ if $t \in \text{post}(s)$, and $\rightarrow^+ (\rightarrow^*)$, for its (reflexive) transitive closure. An accepting cycle is some state $a \in \mathcal{A}$, which is reachable from the initial state ($s_I \rightarrow^* a$) and lies on a non-trivial cycle ($a \rightarrow^+ a$).

```

1 proc ndfs( $s$ )
2   dfs_blue( $s$ )
3   report no cycle
4 proc dfs_red( $s$ )
5    $s.color := red$ 
6   for all  $t$  in post( $s$ ) do
7     if  $t.color = cyan$ 
8       report cycle & exit
9     else if  $t.color = blue$ 
10       dfs_red( $t$ )
11 proc dfs_blue( $s$ )
12    $s.color := cyan$ 
13   for all  $t$  in post( $s$ ) do
14     if  $t.color = cyan$  and ( $s \in \mathcal{A} \vee t \in \mathcal{A}$ )
15       report cycle & exit
16     if  $t.color = white$ 
17       dfs_blue( $t$ )
18   if  $s \in \mathcal{A}$ 
19     dfs_red( $s$ )
20   else
21      $s.color := blue$ 

```

Figure 1: The (sequential) New NDFS algorithm adapted from [19]

2.1 Nested Depth-First Search

The first linear-time algorithm to detect accepting cycles was proposed by Courcoubetis et al. [6] and is referred to as Nested Depth-First Search (NDFS). NDFS also enjoys the on-the-fly property. This means that the algorithm can terminate as soon as a cycle is detected, without the need to visit (or even construct) the whole graph. This makes NDFS very suitable for bug hunting, besides its use for full verification. Various extensions and optimizations to NDFS have been proposed [11, 19, 9]. Alg. 1 most closely resembles *New NDFS* [19].

In Alg. 1, $ndfs(s_i)$ initiates a *blue* DFS from the initial state, so called since explored states are colored blue (we assume that initially all states are white). A newly visited state is first colored *cyan* (“it is on the DFS-stack”), and during backtracking after exploration, it is colored full *blue*. However, if at l.18 the blue DFS backtracks over an accepting state $s \in \mathcal{A}$, then $dfs_red(s)$ is called, which is the nested *red* DFS to determine whether there exists a cycle containing s . As soon as a cyan state is found on l.7, an accepting cycle is reported [11, 19]. *Early cycle detection* is also possible in the blue DFS at l.14,15. Due to early cycle detection, it does not matter that the cyan color of s is overwritten by red at l.5 [13, Sect. 4.4].

NDFS runs in linear time, since each reachable state is visited at most twice, once in the blue DFS and once in a red DFS. The correctness of NDFS essentially depends on the fact that the red DFSs are initiated on accepting states in the post order imposed by the blue DFS. So the red search will never hit another accepting state that is not already red.

2.2 Embarrassing Parallelization: Swarmed NDFS

The inherently DFS nature of the blue search makes NDFS hard to parallelize, since computing the post order is a P-complete problem [18]. One response has been to develop entirely different algorithms based on Breadth-First Search, cf. Sec. 2.6.

Another approach would be to simply run N isolated instances of NDFS (Alg. 1) in parallel, in the hope that this *swarm* of NDFS workers will detect accepting cycles earlier [10, 13]. Local permutations of the post function direct the workers to different regions of the state space, so their search becomes independent. With $post_i^b$ ($post_i^r$) we denote the permutation of successors used in the blue (red) DFS by worker i . Section 4 analyses the expected and actual improvements due to parallel randomized search.

Although Swarmed NDFS is expected to be profitable for bug hunting, it does not show a speedup in

```

1 proc lndfs( $s, N$ )
2   dfs_blue( $s, 1$ ) || .. || dfs_blue( $s, N$ )
3   report no cycle
4 proc dfs_red( $s, i$ )
5    $s.color[i] := pink$ 
6   for all  $t$  in  $post_i^r(s)$  do
7     if  $t.color[i] = cyan$ 
8       report cycle & exit all
9     if  $t.color[i] \neq pink \wedge \neg t.red$ 
10      dfs_red( $t, i$ )
11   if  $s \in \mathcal{A}$ 
12      $s.count := s.count - 1$ 
13     await  $s.count = 0$ 
14      $s.red := true$ 
15 proc dfs_blue( $s, i$ )
16    $allred := true$ 
17    $s.color[i] := cyan$ 
18   for all  $t$  in  $post_i^b(s)$  do
19     if  $t.color[i] = cyan$  and  $(s \in \mathcal{A} \vee t \in \mathcal{A})$ 
20       report cycle & exit all
21     if  $t.color[i] = white \wedge \neg t.red$ 
22       dfs_blue( $t, i$ )
23     if  $\neg t.red$ 
24        $allred := false$ 
25   if  $allred$ 
26      $s.red := true$ 
27   else if  $s \in \mathcal{A}$ 
28      $s.count := s.count + 1$ 
29     dfs_red( $s, i$ )
30    $s.color[i] := blue$ 

```

Figure 2: The LNDFS algorithm, pruning blue and red DFS by a global red color, adapted from [13].

the absence of accepting cycles, in which case all workers have to go through the complete state space. Indeed, the worst-case complexity of all parallel NDFS variations in this paper is $\mathcal{O}(|\rightarrow| \cdot |N|)$, i.e. linear both in the size of the Büchi automaton and in the number of workers.

In order to improve average speedup, some more synchronization between the workers is needed. Note that a naive global sharing of colors between multiple workers would be incorrect, because it would destroy the post-order properties on which NDFS relies. Next, we discuss two recent proposals for sharing information between the NDFS workers.

2.3 LNDFS: Sharing the Red Color Globally

The basic idea behind LNDFS in Alg. 2 is to share information in the backtrack of the red DFSs [13]. A new *pink* color is introduced at l.5 to signify states on the stack of a red DFS, analogous to cyan for a blue DFS. The cyan, blue and pink colors are all local to worker i , but the red color is shared *globally*. On backtracking from the red DFS, states are colored red at l.14. These red states are ignored by *all blue and red* DFSs (l.21,9), thus pruning the search space for all workers i . To improve pruning during the blue search, the amount of red states is even increased by the *allred* extension from [9] (l.16 and l.23-26).

To ensure correctness, it is necessary to synchronize the red coloring of accepting states (see l.13). Otherwise, the algorithm is incorrect for more than two workers (see [13], which provides a correctness proof for $N > 0$ workers). Scalability of the LNDFS algorithm could be hampered by the need for synchronization, but waiting is only needed when multiple workers start a red search from the same accepting state; this does not happen often in practice. Another reason for limited scalability is that work is only pruned when states can be marked red. Despite the *allred* extension, for input graphs with no (or very few) accepting states, all workers still have to traverse the whole graph.

2.4 ENDFS: an Optimistic Approach with Repair Strategy

The basic idea of ENDFS in Alg. 3 [7] is to share both the blue and the red colors globally; only the cyan and pink colors are local per worker. We deviate from the description in [7] by adding a cyan

```

1 proc endfs( $s, N$ )
2   dfs_blue( $s, 1$ ) || .. || dfs_blue( $s, N$ )
3   report no cycle
4 proc dfs_red( $s, i$ )
5    $s.pink[i] := \mathbf{true}$ 
6    $R_i := R_i \cup \{s\}$ 
7   for all  $t$  in  $post_i^r(s)$  do
8     if  $t.cyan[i]$ 
9       report cycle & exit all
10    if  $t \in \mathcal{A} \wedge \neg t.red$ 
11       $t.dangerous := \mathbf{true}$ 
12    if  $\neg t.red \wedge \neg t.pink[i]$ 
13      dfs_red( $s, i$ )
14 proc dfs_blue( $s, i$ )
15    $s.cyan[i] := \mathbf{true}$ 
16   for all  $t$  in  $post_i^b(s)$  do
17     if  $t.cyan[i]$  and  $(s \in \mathcal{A} \vee t \in \mathcal{A})$ 
18       report cycle & exit all
19     if  $\neg t.cyan[i] \wedge \neg t.blue$ 
20       dfs_blue( $t, i$ )
21    $s.cyan[i] := \mathbf{false}$ 
22    $s.blue := \mathbf{true}$ 
23   if  $s \in \mathcal{A}$ 
24      $R_i := \emptyset$ 
25     dfs_red( $s, i$ )
26   for all  $r \in R_i$  do
27     if  $\neg r.dangerous \vee s = r$ 
28        $r.red := \mathbf{true}$ 
29   if  $s.dangerous$ 
30     ndfs( $s, i$ )

```

Figure 3: The optimistic ENDFS algorithm, marking dangerous states, adapted from [7].

stack and early cycle detection as optimizations, because this enables a fair comparison with LNDFS. Consequently, we also renamed the local colors.

Sharing the blue color can lead to problems, as the post-order is not preserved by the algorithm. ENDFS optimistically proceeds, but if it encounters accepting states that are not yet red during the red search, they are marked dangerous at l.11. Eventually, dangerous states are double-checked in a repair stage, by a separate sequential NDFS using worker-local colors only, at l.29-30. Note that for technical reasons, states are not colored red during backtracking, but just collected in the thread-local set R_i at l.6. Only after termination of the red DFS they are made red (provided they are not dangerous) at l.26-28.

Scalability of the ENDFS algorithm could be hampered by the repair stage, because this proceeds sequentially. Also, marking states red occurs relatively late, potentially leading to more duplicate work within the red DFS.

2.5 A Combined Version: New MC-NDFS

We have recapitulated two very recent multi-core NDFS algorithms, which both seem to have their merits and pitfalls. ENDFS, in the end, resorts to a sequential repair strategy, but it avoids some work duplication due to the global blue color. LNDFS does not need a repair strategy, but the blue DFS is only pruned when there are sufficiently many red states, and the algorithm may have to wait for synchronization.

A simple idea suggests itself here: we could combine the two algorithms and try to reconcile their strong points. The idea is simply to run the optimistic algorithm Alg. 3, but when dangerous states are encountered at l.30, we call the parallel algorithm LNDFS (rather than NDFS).

We expect an improved speedup, because using ENDFS ensures good work sharing, even in the absence of accepting states. And using LNDFS parallelizes the repair strategy, avoiding the important sequential bottleneck of ENDFS. In the actual implementation, we also used a simple load balancing strategy: when a worker finishes ENDFS, it starts helping other workers still in their LNDFS repair phase.

2.6 One-Way-Catch-Them-Young with Maximal Accepting Predecessors

In the next section, we will compare the performance of the various NDFS implementations in terms of their absolute timing and speedup behavior. We will also compare them with the current state-of-the-art algorithm in parallel symbolic model checking, OWCTY-MAP [4] by Barnat et al., which is a member of the branch of BFS-based algorithms.

Basically, it extends the *One-Way-Catch-Them-Young* algorithm (OWCTY [21]), with an initialization phase incorporated from the *Maximal-Accepting-Predecessor* algorithm (MAP [5]). In a nutshell, MAP iteratively propagates unique node identifiers to successors. As soon as an accepting state receives its own identifier, a cycle is detected. OWCTY is based on topological sort and iteratively eliminates states that cannot lie on an accepting cycle, because they have no predecessors.

These algorithms are generally based on BFS, which is more easy to parallelize than DFS. However, these algorithms sacrifice linear-time behavior and the on-the-fly property. The resulting combination is linear-time for Büchi automata generated from the class of weak LTL properties, and shows on-the-fly behavior for several cases.

3 Experiments

We implemented multi-core Swarmed NDFS and Alg. 2 and Alg. 3 in the multi-core backend of the LTSMIN model checking tool suite [16, 15, 14].¹ We performed experiments on an AMD Opteron 8356 16-core (4×4 cores) server with 64 GB RAM, running a patched Linux 2.6.32 kernel. All tools were compiled using gcc 4.4.3 in 64-bit mode with high compiler optimizations (-O3).

We measured performance characteristics for all 453 models with properties of the BEEM database [17] and compared the runs with the best known parallel LTL model checking algorithm OWCTY-MAP as implemented in DIVINE 2.5 [2, 3]. In fact, we used the latest release available from the development repository on 23 March 2011, which was close to the 2.5 version, except for a few relevant bug fixes.

Note that OWCTY-MAP has been implemented in DIVINE, whereas all NDFS-based algorithms have been implemented in LTSMIN. This should be taken into account when comparing absolute runtimes. LTSMIN implements a generic interface around the fast implementation of the post function of DIVINE, resulting in sequential runtimes that can be twice as slow. On the other hand, LTSMIN internally uses shared hash tables, which are shown to scale better, at least for reachability [15].

To account for the random nature of the algorithms, all experiments were executed a total of 5 times. The data presented in the following subsections reflect the average over those 5 experiments.

3.1 ENDFS Benchmarks

Evangelista et al. [7] used workload distribution measurements to estimate the scalability of ENDFS. Fig. 5 reflects their estimated speedups (the exact numbers were extracted from [8], which provides experiments for more models, but shows equal numbers to those reported in [7]). Fig. 4 shows the speedups that we obtained by measuring real runtimes of the algorithm.

A comparison with the estimated speedups shows that the trend of the lines has been accurately predicted in most cases. A case by case comparison shows, however, that there is some divergence between the exact numbers: models that scale well in “synthetic” benchmarks of Fig. 5 as, for example, *anderson.6.prop4*, *elevator2.3.prop4*, *leader_election.6.prop2* and *szymanski.4.prop4*,

¹Available on the LTSMIN website: <http://fmt.cs.utwente.nl/tools/ltsmin/>.

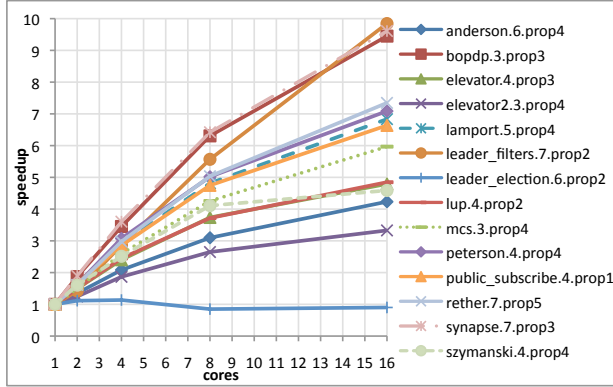


Figure 4: Measured speedups for ENDFS.

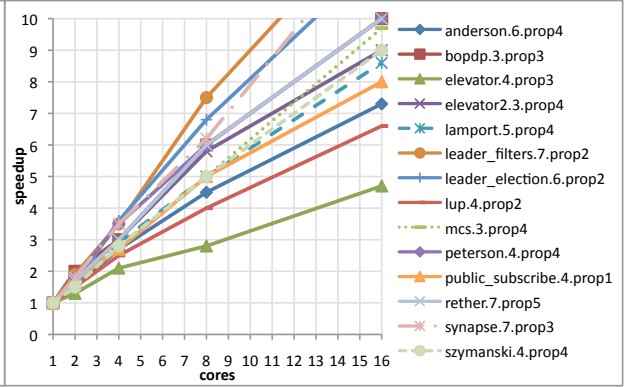


Figure 5: Estimated speedups for ENDFS from [7].

do not scale well in practice. We have not investigated the source of these differences, but apparently the amount of dangerous states is quite sensitive to implementation parameters.

Fig. 6 and Fig. 7 compress the results from all models of the BEEM database in log-log scatter plots. In both figures, we show models without accepting cycles as dots and models with these cycles as crosses. Comparing ENDFS to NDFS in the first figure, we can distinguish good speedups for the models with cycles, while the other figure shows that ENDFS even improves the results of Swarmed NDFS a little. In Section 4, we investigate and compare these effects more thoroughly, using a statistical reference model for random parallel search. As for the models without accepting cycles, we see that most do scale with ENDFS, but hardly beyond a speedup of 10. Even though theoretically possible, we identified no cases where the repair strategy of ENDFS yields speed downs (in the worst case, all workers can traverse the state space 4 times).

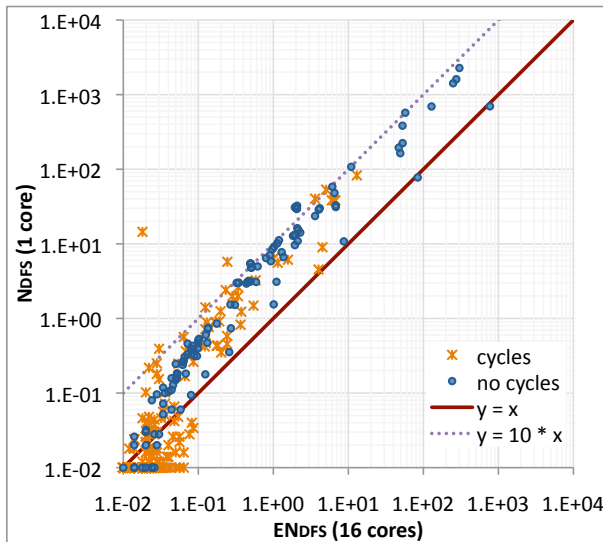


Figure 6: NDFS vs ENDFS.

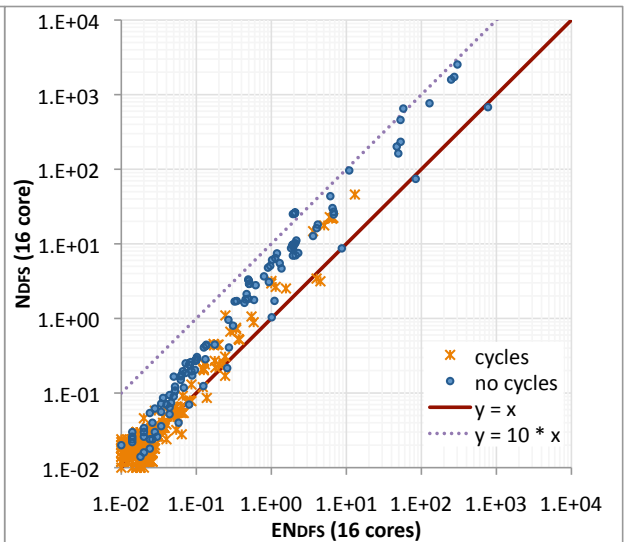


Figure 7: Swarmed NDFS vs ENDFS.

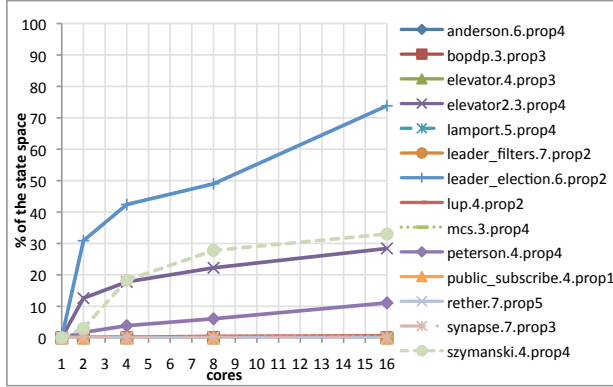


Figure 8: State space coverage of ENDFS repair.

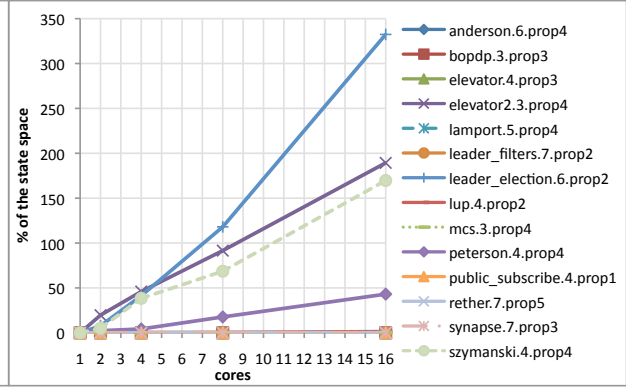


Figure 9: Cumulative extra work due to repair.

We also investigated what caused some inputs to scale poorly. Fig. 8 shows the percentage of the state space that is covered by the repair procedure. As expected, a high percentage was measured for all models with poor scalability. Fig. 9 shows the cumulative additional work performed by all workers, by summing up the states visited by all workers in the repair procedure and dividing by the total amount of states ($|\mathcal{S}|$). It is worrisome that the need for repair can increase faster than the number of cores. This suggests that the ENDFS may not scale to many-core systems.

3.2 ENDFS versus LNDFS

Fig. 10 shows the speedups of the LNDFS algorithm. In this set of models, few scale well with this algorithm. The flat lines represent models with relatively few states reachable from accepting states. In these cases, the algorithm can only color few states red, thus limiting work sharing between the workers. As shown in [13], the fraction of red states is indeed directly related to the speedup that is obtained. The two models `leader_filters.7.prop2` and `leader_election.6.prop2` have state spaces that are colored entirely red, and hence exhibit almost ideal linear speedups. However, Fig. 12 shows that only few models behave this ideally. Unfortunately, in [13] we reported better speedups, which we have

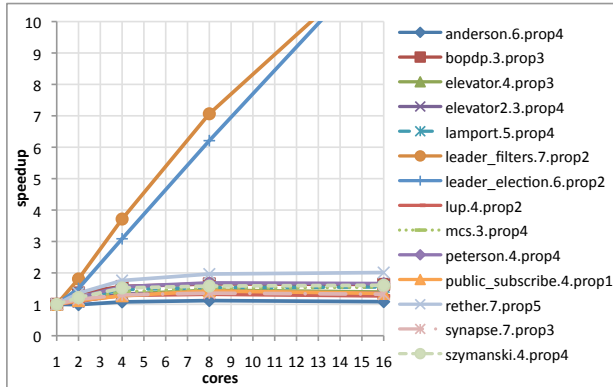


Figure 10: Speedups LNDFS.

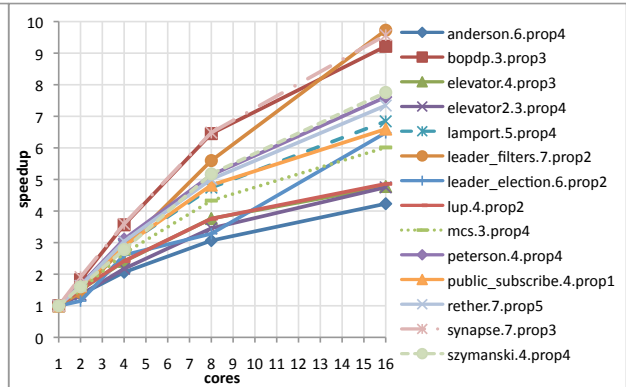


Figure 11: Speedups NMC-NDFS.

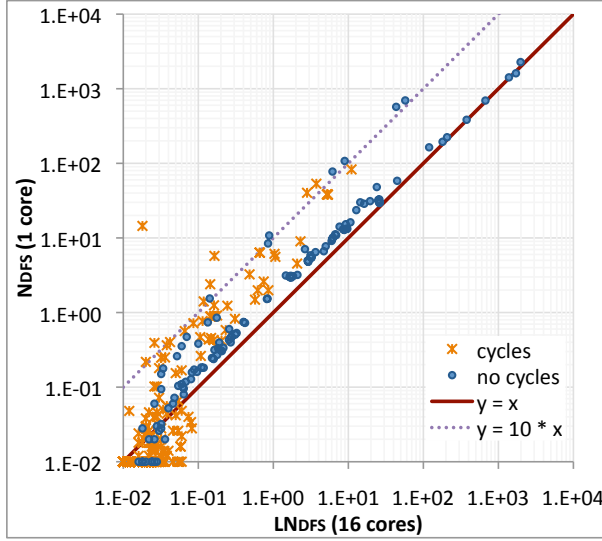


Figure 12: NDFS vs LNDfs.

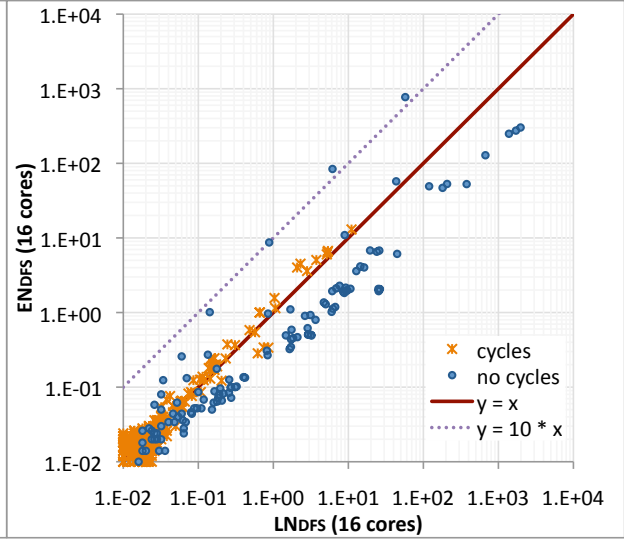


Figure 13: ENDFS vs LNDfs.

now tracked down to an implementation error that led to too many red states.

When comparing ENDFS to LNDfs in Fig. 13, we witness a few ties (on the thick line), a few winners with LNDfs and by far the most winners with ENDFS. We looked up the models that draw a tie and found that all of them scale with both algorithms. These are therefore not in need of improvements. Most interestingly, the models that scale well with LNDfs correspond to those that do not scale with ENDFS. This indicates that both algorithms are complementary. A fact that is indeed to be expected, because the same accepting states that cause states to be colored red in LNDfs, are potentially marked dangerous in ENDFS. This motivated their combination as described in Section 2.5.

3.3 NMC-NDFS Benchmarks

In this subsection, we investigate our proposal for the combination of ENDFS and LNDfs into NMC-NDFS. Fig. 11 shows that NMC-NDFS improves upon the speedups of ENDFS (see Fig. 4), and Fig. 14 confirms that all models scale well with the combined algorithm.

For NMC-NDFS, again, we also calculated the cumulative additional work as a percentage of the state space in Fig. 16. The state space coverage by the repair procedure is almost equal to that of ENDFS in Fig. 8. We can then deduce that the repair work is parallelized well by LNDfs, because the cumulative additional work is close to the percentage of state space coverage. This can be explained by the fact that LNDfs is always called on a (dangerous) accepting state in NMC-NDFS, which eventually leads to a red coloring of the entire subgraph reachable from this accepting state. Under these conditions LNDfs can be expected to scale well.

We also checked whether the new combination causes additional overhead, by comparing it directly with its predecessors in Fig. 18 and Fig. 19. The first figure shows that no model runs faster with ENDFS than with NMC-NDFS, although in a few examples LNDfs wins, as can be seen in the latter figure. This confirms that LNDfs and ENDFS are complementary and their combination represents the best from both worlds. Indeed, the combination ensures that for all inputs some speedup is obtained.

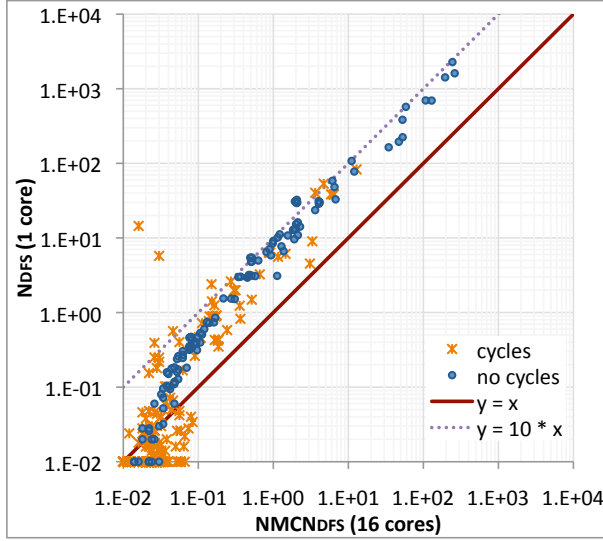


Figure 14: NDFS vs NMC-NDFS.

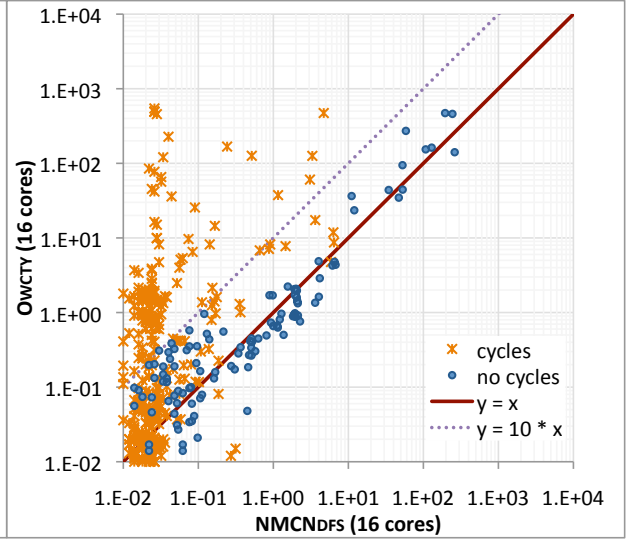


Figure 15: OWCTY-MAP vs NMC-NDFS.

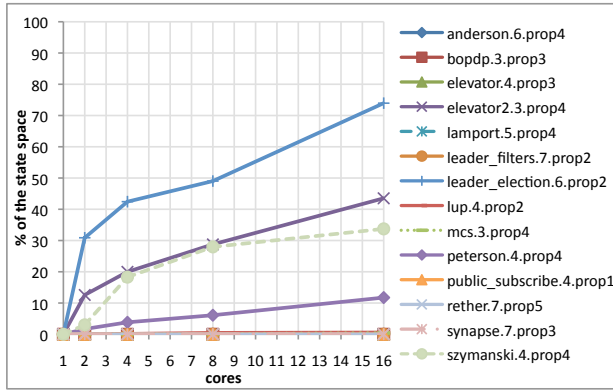


Figure 16: Cumulative extra work due to NMC-NDFS repair.

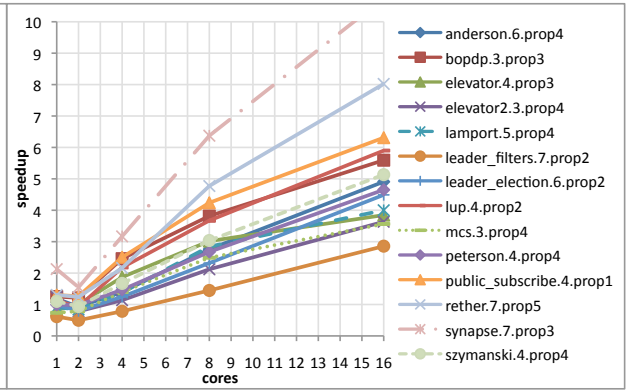


Figure 17: Speedups OWCTY.

3.4 Parallel NDFS versus OWCTY-MAP

Fig. 15 compares NMC-NDFS with OWCTY-MAP. The comparison figures show that the heuristic on-the-fly method of OWCTY-MAP is no match for the truly on-the-fly parallel NDFS algorithms. As for the models without accepting cycles, we can conclude that currently NMC-NDFS provides a good match for OWCTY-MAP, in particular for the larger models. For the sake of completeness, we present here Fig. 20, 21, which show a comparison between ENDFS/LNDFS and OWCTY-MAP. Furthermore, Fig. 17 shows the absolute speedups of OWCTY-MAP using the sequential NDFS runtimes as the base case.

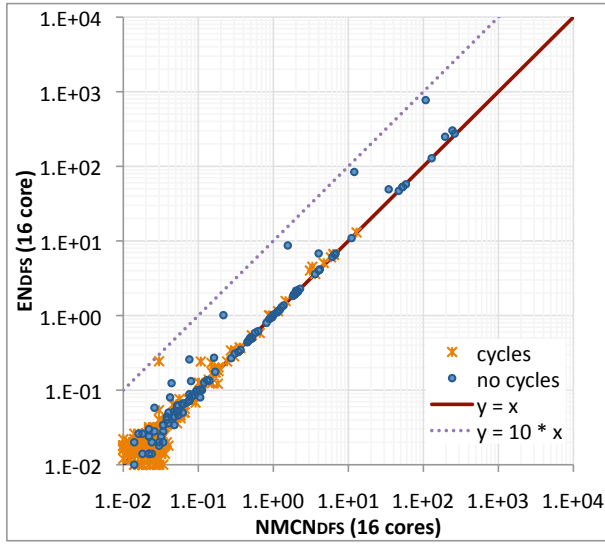


Figure 18: ENDFS vs NMC-NDFS.

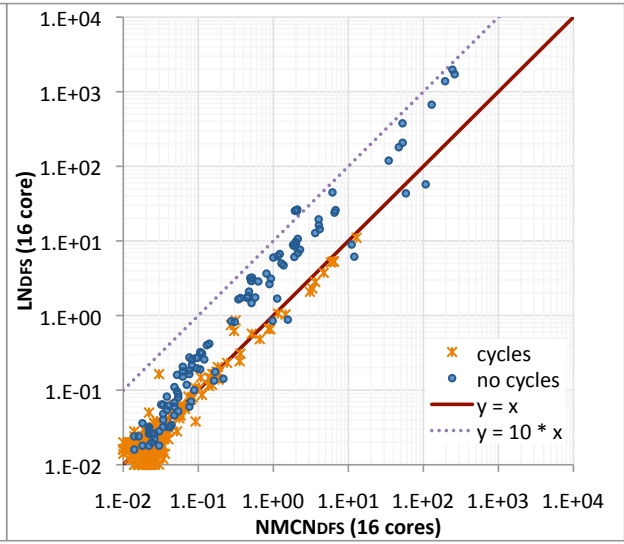


Figure 19: LNDFS vs NMC-NDFS.

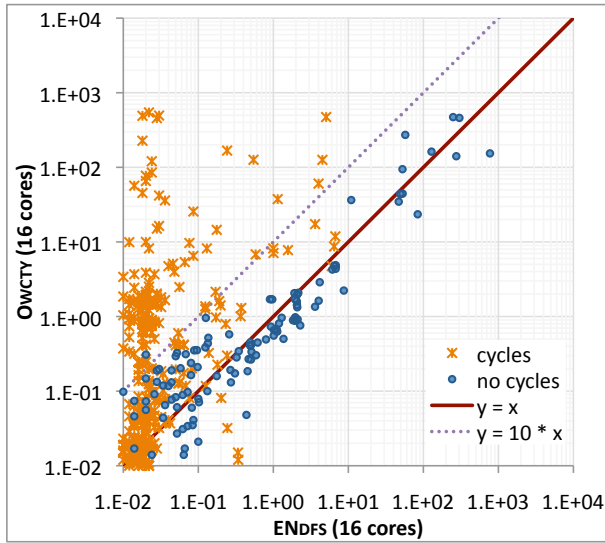


Figure 20: OWCTY-MAP vs ENDFS.

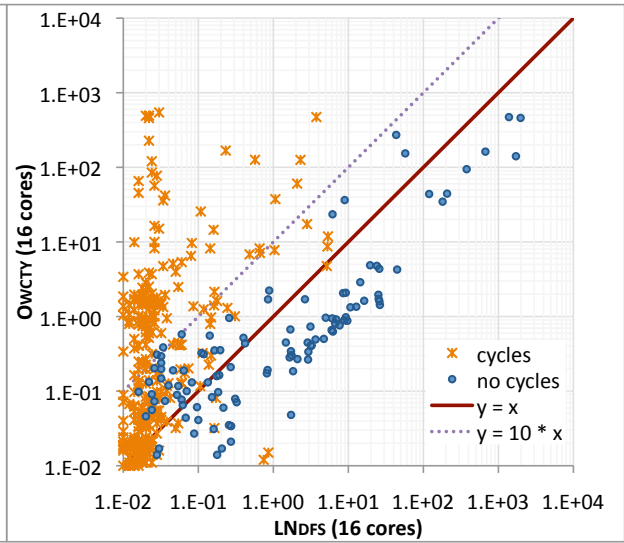


Figure 21: OWCTY-MAP vs LNDFS.

4 Discussion on Parallel Random Search

As explained in Section 2, the multi-core NDFS algorithms use a randomized post function to direct workers to different regions of the state space. In this section, we want to explain the speedup for models with accepting cycles. In particular, we want to distinguish the effect of parallel random search, from the effect of the clever work sharing algorithms.

Our starting point is a simple statistical model as found in [12]. We view $\text{NDFS}(\mathcal{B}, X)$ as an algorithm that runs on Büchi automaton \mathcal{B} with random seed X , influencing the order of traversing successors. We ran $\text{NDFS}(\mathcal{B}, X)$ 500 times with random X on a number of Büchi automata \mathcal{B} . Each time, we measured $f(\mathcal{B}, X)$, the time that it takes for $\text{NDFS}(\mathcal{B}, X)$ to detect an accepting cycle.

In Figure 22, we show the cumulative probability $F(\mathcal{B}, t)$ that one NDFS worker will detect an accepting cycle in less than t seconds for some examples from the BEEM database. We can also define $F_N(\mathcal{B}, t)$ as the cumulative probability that a swarm of N independent workers will find an accepting cycle within t seconds. Figure 23 shows $F_{16}(\mathcal{B}, t)$ for the same automata. We also computed the expected time to completion and the standard deviation. The new distribution can be easily computed as:

$$F_N(\mathcal{B}, t) = 1 - (1 - F(\mathcal{B}, t))^N$$

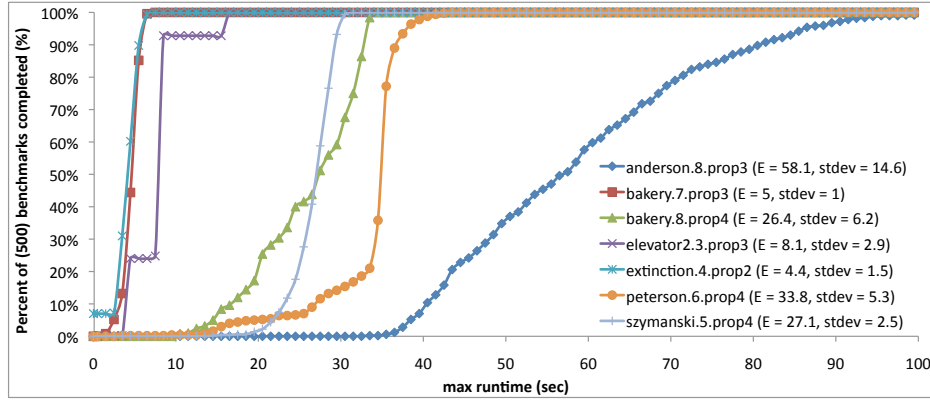


Figure 22: Cumulative probability distribution of finding a bug (measured for 1 worker).

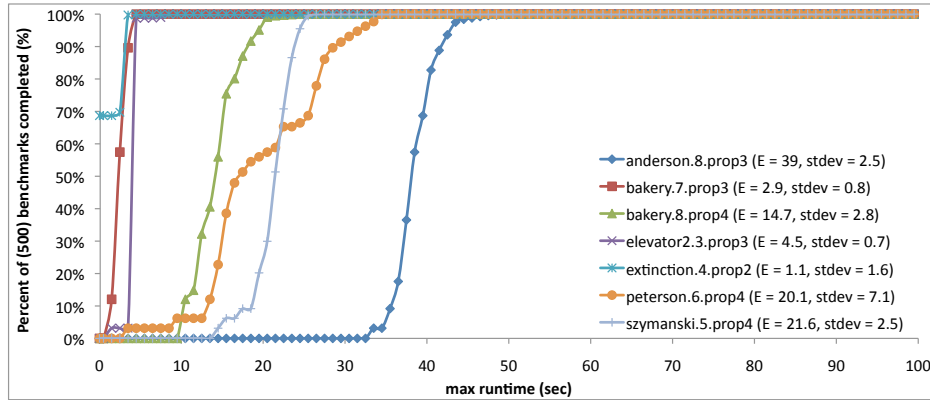


Figure 23: Cumulative probability distribution of finding a bug (calculated for 16 workers).

From Fig. 22 and Fig. 23, we observe that considerable gains can be expected from a simple parallelization as in Swarmed NDFS. It also shows that the actual speedup depends highly on the models: when all runs find an accepting cycle in about the same time (indicated by plateaus connected by a steep curve), the expected gain is much less than when the curve is flatter, as is the case for `anderson.8.prop3`, `bakery.8.prop4` and `peterson.6.prop4`.

Next, we want to compare our actual implementation with these predictions. To this end, we compared the expected completion times with actual completion times, averaged over 5 runs. We collected this information in Table 1. In the first two columns (Statistical model), we copied the averages from Fig. 22, 23 for 1 and 16 workers, and computed the expected speedup. Note that this speedup for 16 workers is way below 16. Next, we experimented with four different scenarios described below.

The next column (Distributed), corresponds to Swarmed NDFS as it would run on different machines in a GRID. Here the only synchronization would be to terminate all workers as soon as the first worker has detected a cycle. The runtimes denote the completion time for the earliest run out of 16 independent workers; we again provide the average from 5 experiments. The corresponding speedups match closely to the predicted ones from the statistical model.

Table 1: Runtimes and speedups of bug hunting using embarrassingly parallel (randomized) NDFS and LNDFS. The first two columns of the table present the expected completion time derived from 500 sequential experiments for 1 and 16 cores. The other columns give parallel runtimes for, respectively, a distributed implementation, our randomized shared-memory implementation [13], and another shared-memory implementation using the fresh successor heuristic. The second row gives the speedups.

		NDFS					LNDFS	
		1 core	16 core				16 core	
	model	Statistical model	Statistical model	Distributed	Shared Memory	Heuristic	Shared Memory	Heuristic
Runtimes (sec)	<code>anderson.8.prop3</code>	58.1	39.0	39.3	39.4	9.6	8.6	3.1
	<code>bakery.7.prop3</code>	5.0	2.9	2.9	2.1	0.6	0.8	0.3
	<code>bakery.8.prop4</code>	26.4	14.7	13.6	12.9	0.6	1.9	1.1
	<code>elevator2.3.prop3</code>	8.1	4.5	4.2	2.6	0.7	2.1	0.2
	<code>extinction.4.prop2</code>	4.4	1.1	0.8	0.5	0.0	0.0	0.0
	<code>peterson.6.prop4</code>	33.8	20.1	24.2	16.7	12.5	2.5	2.2
	<code>szymanski.5.prop4</code>	27.1	21.6	20.9	19.4	0.0	3.3	0.0
Speedups	<code>anderson.8.prop3</code>		1.5	1.5	1.5	6.1	6.7	18.5
	<code>bakery.7.prop3</code>		1.7	1.7	2.4	8.6	6.3	15.2
	<code>bakery.8.prop4</code>		1.8	1.9	2.0	45.7	14.1	23.2
	<code>elevator2.3.prop3</code>		1.8	1.9	3.1	11.7	3.8	41.8
	<code>extinction.4.prop2</code>		4.1	5.9	8.9	??	??	??
	<code>peterson.6.prop4</code>		1.7	1.4	2.0	2.7	13.5	15.6
	<code>szymanski.5.prop4</code>		1.3	1.3	1.4	??	8.3	??

Next, we ran the experiments on the multi-core machine with 16 cores described before. Now the workers share the basic infrastructure. This is the same setting as the multi-core Swarmed NDFS from the previous section. For instance, all states will be stored only once in a shared hash table. Also, several workers now share information in the L2 cache. On the other hand, they might now suffer from cache coherence overhead or memory bus contention. The figures under “Shared Memory” show that the speedups in a multi-core environment are slightly better than on independent machines (Distributed).

On multi-core machines it becomes easier to share information, in order to guide different workers into different parts of the state space. In that case, one would expect better speedup figures. We did an experiment with what we call the *fresh successor heuristic*. Here a worker will randomly select a globally unvisited successor if that exists, otherwise it randomly selects any successor. As the column Heuristic shows, this can dramatically improve the speedup of 16 workers. In some cases, each time an accepting cycle was found in such a small instant that a meaningful speedup figure could not be computed.

Finally, using LNDFS, the total amount of work is decreased, because workers prune each other’s search space. Again, we experimented with two versions, which are shown in the two right-most columns. We computed the average runtime of 5 experiments on 16 cores with the random shared-memory implementation. Note that this is the implementation that was used in all previous experiments in Section 3. The figures show again a big improvement over Swarmed NDFS, even on a multi-core machine. Interestingly, the fresh successor heuristic also works very well for the LNDFS-algorithm, speeding up the algorithm several times. Similar findings hold for all other parallel NDFS versions in this paper, because they behave similarly on models with accepting cycles (see Fig. 18 and Fig. 19).

5 Conclusion

In this paper, we experimentally compared two recent parallel NDFS-based algorithms, ENDFS [7] and LNDFS [13]. We also compared them with Swarmed NDFS and with the BFS-based algorithm OWCTY-MAP [4]. We now summarize the conclusions from our experiments.

For systems with bugs (accepting cycles), both ENDFS and LNDFS outperform OWCTY-MAP by large, so they fully enjoy the on-the-fly property. We have also shown that for these cases ENDFS and LNDFS perform much better than parallel random search, as in Swarmed NDFS.

On examples without bugs, it appears that ENDFS beats LNDFS in most of the cases, due to the fact that there are still too few red states to prune the blue search in LNDFS. However, in a number of other cases ENDFS still scales rather badly, due to the fact that the sequential repair strategy traverses large parts of the state space. Interestingly, it is possible to use the parallel LNDFS algorithm as the repair strategy of ENDFS. For this new combined algorithm, all examples of the BEEM database showed a decent speedup.

On examples without bugs, OWCTY-MAP beats both LNDFS and ENDFS in a majority of the cases, but still it is slower on a number of other examples. The combination of ENDFS and LNDFS, however, provided a good match for OWCTY-MAP, especially for the larger inputs. This shows that the new branch of parallel NDFS algorithms is rather promising.

Future work. We believe that the last word on parallel LTL model checking has not been said yet. Although all NDFS-versions have been implemented in the same framework so that we compare the algorithmic differences, OWCTY-MAP was implemented in the DiViNE tool. We note that our computation of the post function uses the same code from DiViNE. A reimplementing of OWCTY-MAP using shared hash tables will probably increase its speedup, as indicated by results on pure reachability [15].

Also the young branch of parallel NDFS algorithms can still be improved. We have already shown that adding heuristics to direct workers into different regions of the graph can greatly increase the performance, at least for models with bugs. An interesting question is if there exists a correct variation on parallel NDFS that can fully share global information from both the blue and the red search, without the need to resort to a repair strategy. This would take away the current weak points of both ENDFS and LNDFS.

Acknowledgement. We thank Jiří Barnat and Keijo Heljanko for organizing PDMC 2011 and inviting our contribution in this volume. Moreover, we thank Jiří Barnat for his comments on the time complexity of parallel randomized algorithms, and Keijo Heljanko for his pointer to the statistical model in [12]. Finally, we are grateful to Mark Timmer for his help on the statistics.

References

- [1] C. Baier & J.P. Katoen (2008): *Principles of Model Checking*. The MIT Press.
- [2] J. Barnat, L. Brim & P. Ročkal (2010): *Scalable Shared Memory LTL Model Checking*. *STTT* 12(2), pp. 139–153, doi:10.1007/s10009-010-0136-z.
- [3] J. Barnat, L. Brim, M. Češka & P. Ročkal (2010): *DiVinE: Parallel Distributed Model Checker (Tool paper)*. In: *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC 2010)*, IEEE, pp. 4–7, doi:10.1007/978-3-540-88387-6_20.
- [4] L. Barnat, L. Brim & P. Ročkal (2009): *A Time-Optimal On-The-Fly Parallel Algorithm for Model Checking of Weak LTL Properties*. In: *ICFEM 2009, LNCS 5885*, Springer, Heidelberg, pp. 407–425, doi:10.1007/978-3-642-10373-5_21.
- [5] L. Brim, I. Cerná, P. Moravec & J. Simsa (2004): *Accepting Predecessors Are Better than Back Edges in Distributed LTL Model-Checking*. In A.J. Hu & A.K. Martin, editors: *FMCAD, Lecture Notes in Computer Science* 3312, Springer, pp. 352–366, doi:10.1007/978-3-540-30494-4_25.
- [6] C. Courcoubetis, M.Y. Vardi, P. Wolper & M. Yannakakis (1992): *Memory-Efficient Algorithms for the Verification of Temporal Properties*. *Formal Methods in System Design* 1(2/3), pp. 275–288, doi:10.1007/BFb0023737.
- [7] S. Evangelista, L. Petrucci & S. Youcef (2011): *Parallel Nested Depth-First Searches for LTL Model Checking*. In T. Bultan & P.-A. Hsiung, editors: *Automated Technology for Verification and Analysis 2011, Lecture Notes in Computer Science* 6996, Springer, pp. 381–396, doi:10.1007/978-3-642-24372-1_27.
- [8] S. Evangelista, L. Petrucci & S. Youcef (Last accessed 15 Sept 2011): *Parallel Nested Depth-First Searches for LTL Model Checking*. Technical Report, Université Paris 13. Available at <http://www-lipn.univ-paris13.fr/~evangelista/doc/mc-ndfs.pdf>.
- [9] A. Gaiser & S. Schwoon (2009): *Comparison of Algorithms for Checking Emptiness on Büchi Automata*. In P. Hliněný, V. Matyáš & T. Vojnar, editors: *MEMICS’09, OpenAccess Series in Informatics (OASICS)* 13, Schloss Dagstuhl, Germany, doi:10.4230/DROPS.MEMICS.2009.2349.
- [10] G.J. Holzmann, R. Joshi & A. Groce (2008): *Swarm Verification*. In: *ASE, IEEE, L’Aquila, Italy*, pp. 1–6, doi:10.1109/ASE.2008.9.
- [11] G.J. Holzmann, D. Peled & M. Yannakakis (1996): *On Nested Depth First Search*. In: *The SPIN Verification System*, American Mathematical Society, pp. 23–32.
- [12] A.E.J. Hyvärinen, T.A. Junttila & I. Niemelä (2008): *Strategies for Solving SAT in Grids by Randomized Search*. In S. Autexier, J. Campbell, J. Rubio, V. Sorge, M. Suzuki & F. Wiedijk, editors: *AISC/MKM/Calculus*, LNCS 5144, Springer, pp. 125–140, doi:10.1007/978-3-540-85110-3_11.

- [13] A.W. Laarman, R. Langerak, J.C. van de Pol, M. Weber & A. Wijs (2011): *Multi-Core Nested Depth-First Search*. In T. Bultan & P.-A. Hsiung, editors: *Automated Technology for Verification and Analysis 2011, Lecture Notes in Computer Science* 6996, Springer, pp. 321–335, doi:10.1007/978-3-642-24372-1_23.
- [14] A.W. Laarman, J.C. van de Pol & M. Weber (2011): *Parallel Recursive State Compression for Free*. In A. Groce & M. Musuvathi, editors: *SPIN, Lecture Notes in Computer Science* 6823, Springer, Snowbird, USA, pp. 38–56, doi:10.1007/978-3-642-22306-8_4.
- [15] A.W. Laarman, J.C. van de Pol & M. Weber (2010): *Boosting Multi-Core Reachability Performance with Shared Hash Tables*. In N. Sharygina & R. Bloem, editors: *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design, Lugano, Swiss, IEEE Computer Society, USA*, pp. 247–256. Available at <http://eprints.eemcs.utwente.nl/18437/>.
- [16] A.W. Laarman, J.C. van de Pol & M. Weber (2011): *Multi-Core LTSmin: Marrying Modularity and Scalability*. In M. Bobaru, K. Havelund, G. Holzmann & R. Joshi, editors: *Proceedings of the Third International Symposium on NASA Formal Methods, NFM 2011, Pasadena, CA, USA, LNCS* 6617, Springer Verlag, Berlin, pp. 506–511, doi:10.1007/978-3-642-20398-5_40.
- [17] R. Pelánek (2007): *BEEM: Benchmarks for Explicit Model Checkers*. In: *Proc. of SPIN Workshop, LNCS* 4595, Springer, pp. 263–267, doi:10.1007/978-3-540-73370-6_17.
- [18] J.H. Reif (1985): *Depth-first Search is Inherently Sequential*. *Information Processing Letters* 20(5), pp. 229–234, doi:10.1016/0020-0190(85)90024-9.
- [19] S. Schwoon & J. Esparza (2005): *A Note on On-the-Fly Verification Algorithms*. In Nicolas Halbwachs & Lenore D. Zuck, editors: *Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science* 3440, Springer, pp. 174–190, doi:10.1007/978-3-540-31980-1_12.
- [20] M.Y. Vardi & P. Wolper (1986): *An Automata-Theoretic Approach to Automatic Program Verification*. In: *Proc. 1st Symp. on Logic in Computer Science*, Cambridge, pp. 332–344. Available at <http://www.cs.rice.edu/~vardi/papers/lics86.pdf.gz>.
- [21] I. Černá & R. Pelánek (2003): *Distributed Explicit Fair Cycle Detection (Set Based Approach)*. In T. Ball & S.K. Rajamani, editors: *SPIN, Lecture Notes in Computer Science* 2648, Springer, pp. 49–73, doi:10.1007/3-540-44829-2_4.